Table 5-55
VHDL program that allows adder sharing.

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

entity vaddshr is
    port (
        A, B, C, D: in SIGNED (7 downto 0);
        SEL: in STD_LOGIC;
        S: out SIGNED (7 downto 0)
    );
end vaddshr;

architecture vaddshr_arch of vaddshr is
begin
    S <= A + B when SEL = '1' else C + D;
end vaddshr_arch;
```

one's output with a multiplexer, the synthesis engine can build just one adder and select its inputs using multiplexers, potentially creating a smaller overall circuit.

*5.11 Combinational Multipliers

*5.11.1 Combinational Multiplier Structures

combinational multiplier

In Section 2.8, we outlined an algorithm that uses n shifts and adds to multiply n-bit binary numbers. Although the shift-and-add algorithm emulates the way that we do paper-and-pencil multiplication of decimal numbers, there is nothing inherently "sequential" or "time dependent" about multiplication. That is, given two n-bit input words X and Y, it is possible to write a truth table that expresses the $2n$-bit product $P = X \cdot Y$ as a *combinational* function of X and Y. A *combinational multiplier* is a logic circuit with such a truth table.

product component

Most approaches to combinational multiplication are based on the paper-and-pencil shift-and-add algorithm. Figure 5-96 illustrates the basic idea for an 8×8 multiplier for two unsigned integers, multiplicand $X = x_7x_6x_5x_4x_3x_2x_1x_0$ and multiplier $Y = y_7y_6y_5y_4y_3y_2y_1y_0$. We call each row a *product component*, a shifted

Figure 5-96
Partial products in an 8×8 multiplier.

									y_0x_7	y_0x_6	y_0x_5	y_0x_4	y_0x_3	y_0x_2	y_0x_1	y_0x_0
								y_1x_7	y_1x_6	y_1x_5	y_1x_4	y_1x_3	y_1x_2	y_1x_1	y_1x_0	
							y_2x_7	y_2x_6	y_2x_5	y_2x_4	y_2x_3	y_2x_2	y_2x_1	y_2x_0		
						y_3x_7	y_3x_6	y_3x_5	y_3x_4	y_3x_3	y_3x_2	y_3x_1	y_3x_0			
					y_4x_7	y_4x_6	y_4x_5	y_4x_4	y_4x_3	y_4x_2	y_4x_1	y_4x_0				
				y_5x_7	y_5x_6	y_5x_5	y_5x_4	y_5x_3	y_5x_2	y_5x_1	y_5x_0					
			y_6x_7	y_6x_6	y_6x_5	y_6x_4	y_6x_3	y_6x_2	y_6x_1	y_6x_0						
+		y_7x_7	y_7x_6	y_7x_5	y_7x_4	y_7x_3	y_7x_2	y_7x_1	y_7x_0							
	p_{15}	p_{14}	p_{13}	p_{12}	p_{11}	p_{10}	p_9	p_8	p_7	p_6	p_5	p_4	p_3	p_2	p_1	p_0

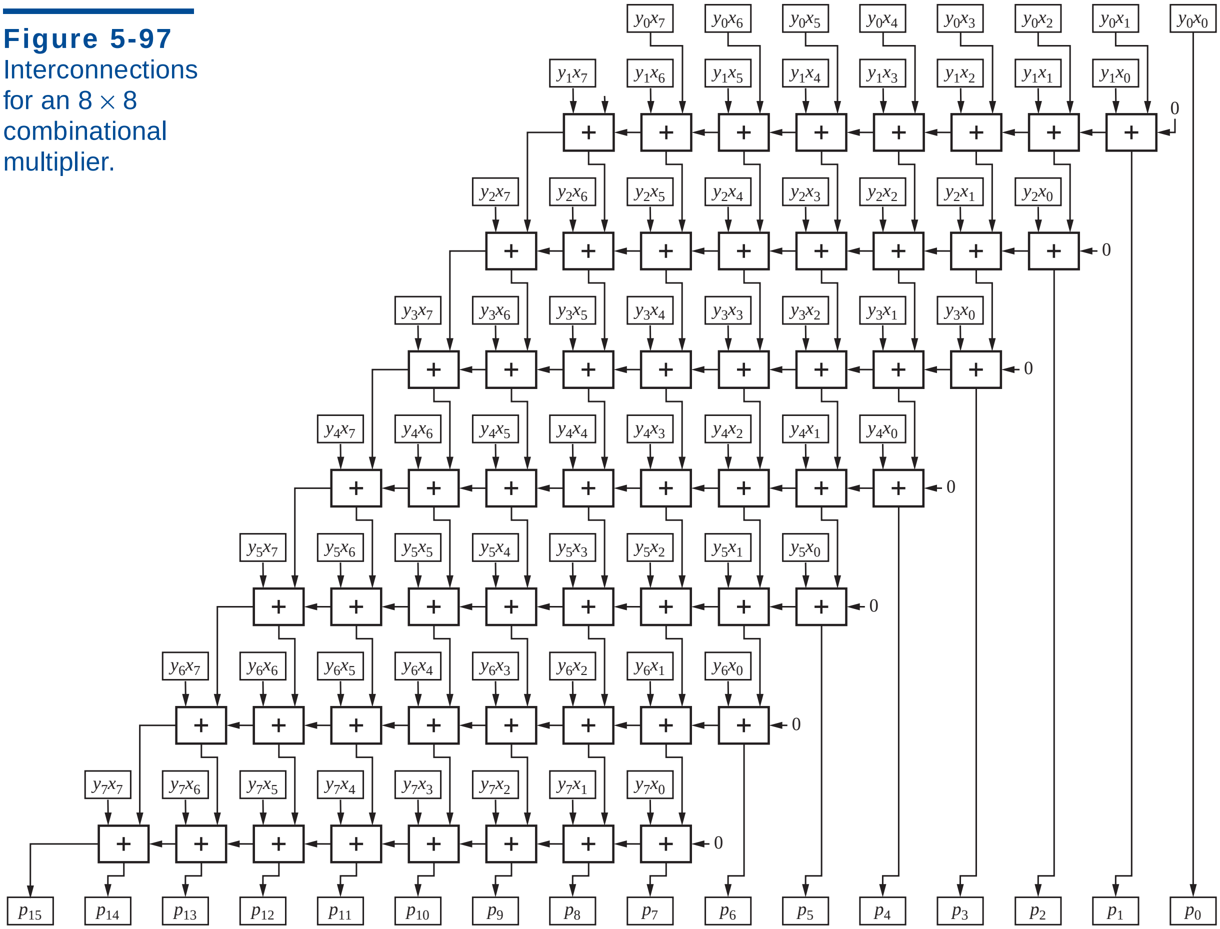

Figure 5-97 Interconnections for an 8 × 8 combinational multiplier.

multiplicand that is multiplied by 0 or 1 depending on the corresponding multiplier bit. Each small box represents one product-component bit y_ix_j, the logical AND of multiplier bit y_i and multiplicand bit x_j. The product $P = p_{15}p_{14}\ldots p_2p_1p_0$ has 16 bits and is obtained by adding together all the product components.

Figure 5-97 shows one way to add up the product components. Here, the product-component bits have been spread out to make space, and each "+" box is a full adder equivalent to Figure 5-85(c) on page 391. The carries in each row of full adders are connected to make an 8-bit ripple adder. Thus, the first ripple adder combines the first two product components to product the first partial product, as defined in Section 2.8. Subsequent adders combine each partial product with the next product component.

It is interesting to study the propagation delay of the circuit in Figure 5-97. In the worst case, the inputs to the least significant adder (y_0x_1 and y_1x_0) can affect the MSB of the product (p_{15}). If we assume for simplicity that the delays from any input to any output of a full adder are equal, say t_{pd}, then the worst case

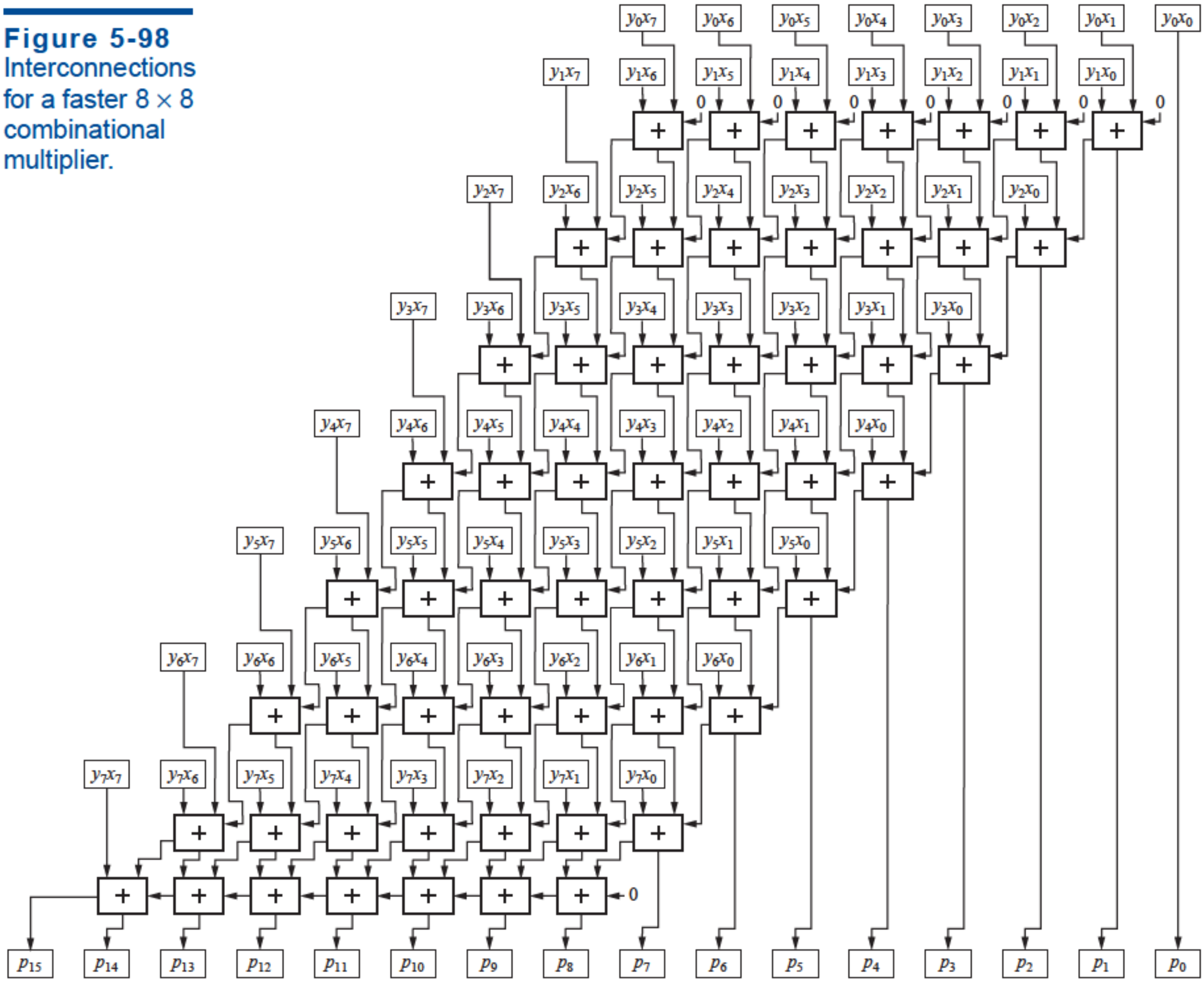

Figure 5-98 Interconnections for a faster 8×8 combinational multiplier.

path goes through 20 adders and its delay is $20t_{pd}$. If the delays are different, then the answer depends on the relative delays; see Exercise \exref{xxxx}.

sequential multiplier

Sequential multipliers use a single adder and a register to accumulate the partial products. The partial-product register is initialized to the first product component, and for an $n \times n$-bit multiplication, $n-1$ steps are taken and the adder is used $n-1$ times, once for each of the remaining $n-1$ product components to be added to the partial-product register.

carry-save addition

Some sequential multipliers use a trick called *carry-save addition* to speed up multiplication. The idea is to break the carry chain of the ripple adder to shorten the delay of each addition. This is done by applying the carry output from bit i during step j to the carry input for bit $i+1$ during the *next* step, $j+1$. After the last product component is added, one more step is needed in which the carries are hooked up in the usual way and allowed to ripple from the least to the most significant bit.

The combinational equivalent of an 8×8 multiplier using carry-save addition is shown in Figure 5-98. Notice that the carry out of each full adder in the first seven rows is connected to an input of an adder *below* it. Carries in the eighth row of full adders are connected to create a conventional ripple adder. Although this adder uses exactly the same amount of logic as the previous one (64 2-input AND gates and 56 full adders), its propagation delay is substantially shorter. Its worst-case delay path goes through only 14 full adders. The delay can be further improved by using a carry lookahead adder for the last row.

The regular structure of combinational multipliers make them ideal for VLSI and ASIC realization. The importance of fast multiplication in microprocessors, digital video, and many other applications has led to much study and development of even better structures and circuits for combinational multipliers; see the References.

*5.11.2 Multiplication in ABEL and PLDs

ABEL provides a multiplication operator `*`, but it can be used only with individual signals, numbers, or special constants, not with sets. Thus, ABEL cannot synthesize a multiplier circuit from a single equation like "`P = X*Y`."

Still, you can use ABEL to specify a combinational multiplier if you break it down into smaller pieces. For example, Table 5-56 shows the design of a 4×4 unsigned multiplier following the same general structure as Figure 5-96 on page page 406. Expressions are used to define the four product components, `PC1`, `PC2`, `PC3`, and `PC4`, which are then added in the `equations` section of the program. This does not generate an array of full adders as in Figure 5-97 or 5-98. Rather, the ABEL compiler will dutifully crunch the addition equation to pro-

```
module mul4x4
title '4x4 Combinational Multiplier'

X3..X0, Y3..Y0 pin;  " multiplicand, multiplier
P7..P0         pin istype 'com';    " product

P = [P7..P0];
PC1 = Y0 & [0, 0, 0, 0,X3,X2,X1,X0];
PC2 = Y1 & [0, 0, 0,X3,X2,X1,X0, 0];
PC3 = Y2 & [0, 0,X3,X2,X1,X0, 0, 0];
PC4 = Y3 & [0,X3,X2,X1,X0, 0, 0, 0];

equations
P = PC1 + PC2 + PC3 + PC4;

end mul4x4
```

Table 5-56
ABEL program for a 4×4 combinational multiplier.

duce a minimal sum for each of the eight product output bits. Surprisingly, the worst-case output, P4, has only 36 product terms, a little high but certainly realizable in two passes through a PLD.

*5.11.3 Multiplication in VHDL

VHDL is rich enough to express multiplication in a number of different ways; we'll save the best for last.

Table 5-57 is a behavioral VHDL program that mimics the multiplier structure of Figure 5-98. In order to represent the internal signals in the figure, the program defines a new data type, `array8x8`, which is a two-dimensional array of `STD_LOGIC` (recall that `STD_LOGIC_VECTOR` is a one-dimensional array of `STD_LOGIC`). Variable `PC` is declared as a such an array to hold the product-component bits, and variables `PCS` and `PCC` are similar arrays to hold the sum and carry outputs of the main array of full adders. One-dimensional arrays `RAS` and `RAC` hold the sum and carry outputs of the ripple adder. Figure 5-99 shows the variable naming and numbering scheme. Integer variables `i` and `j` are used as loop indices for rows and columns, respectively.

The program attempts to illustrate the logic gates that would be used in a faithful realization of Figure 5-98, even though a synthesizer could legitimately create quite a different structure from this behavioral program. If you want to control the structure, then you must use structural VHDL, as we'll show later.

In the program, the first, nested `for` statement performs 64 `AND` operations to obtain the product-component bits. The next `for` loop initializes boundary conditions at the top of the multiplier, using the notion of row-0 "virtual" full adders, not shown in the figure, whose sum outputs equal the first row of `PC` bits

Figure 5-99
VHDL variable names for the 8 × 8 multiplier.

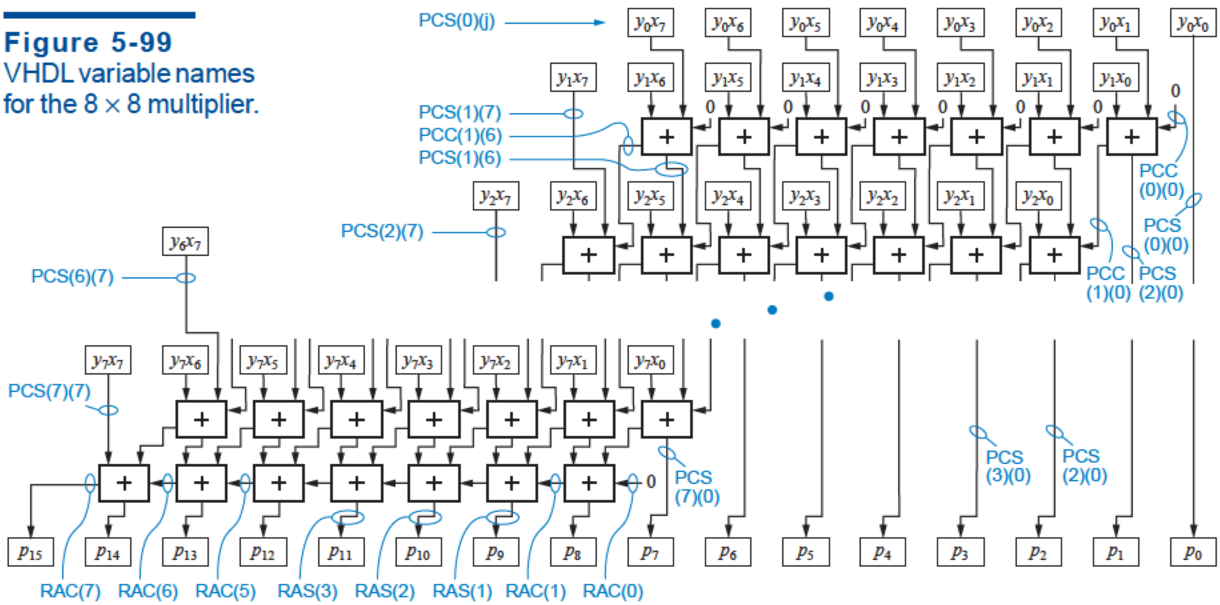

Table 5-57 Behavioral VHDL program for an 8×8 combinational multiplier.

```
library IEEE;
use IEEE.std_logic_1164.all;

entity vmul8x8p is
    port ( X: in STD_LOGIC_VECTOR (7 downto 0);
           Y: in STD_LOGIC_VECTOR (7 downto 0);
           P: out STD_LOGIC_VECTOR (15 downto 0) );
end vmul8x8p;

architecture vmul8x8p_arch of vmul8x8p is
function MAJ (I1, I2, I3: STD_LOGIC) return STD_LOGIC is
  begin
    return ((I1 and I2) or (I1 and I3) or (I2 and I3));
  end MAJ;
begin
process (X, Y)
type array8x8 is array (0 to 7) of STD_LOGIC_VECTOR (7 downto 0);
variable PC: array8x8;     -- product component bits
variable PCS: array8x8;    -- full-adder sum bits
variable PCC: array8x8;    -- full-adder carry output bits
variable RAS, RAC: STD_LOGIC_VECTOR (7 downto 0); -- ripple adder sum
  begin                                           --   and carry bits
    for i in 0 to 7 loop for j in 0 to 7 loop
        PC(i)(j) := Y(i) and X(j); -- compute product component bits
    end loop;  end loop;
    for j in 0 to 7 loop
      PCS(0)(j) := PC(0)(j);  -- initialize first-row "virtual"
      PCC(0)(j) := '0';       --   adders (not shown in figure)
    end loop;
    for i in 1 to 7 loop      -- do all full adders except last row
      for j in 0 to 6 loop
        PCS(i)(j) := PC(i)(j) xor PCS(i-1)(j+1) xor PCC(i-1)(j);
        PCC(i)(j) := MAJ(PC(i)(j), PCS(i-1)(j+1), PCC(i-1)(j));
        PCS(i)(7) := PC(i)(7); -- leftmost "virtual" adder sum output
      end loop;
    end loop;
    RAC(0) := '0';
    for i in 0 to 6 loop  -- final ripple adder
      RAS(i) := PCS(7)(i+1) xor PCC(7)(i) xor RAC(i);
      RAC(i+1) := MAJ(PCS(7)(i+1), PCC(7)(i), RAC(i));
    end loop;
    for i in 0 to 7 loop
      P(i) <= PCS(i)(0);  -- first 8 product bits from full-adder sums
    end loop;
    for i in 8 to 14 loop
      P(i) <= RAS(i-8);   -- next 7 bits from ripple-adder sums
    end loop;
    P(15) <= RAC(7);      -- last bit from ripple-adder carry
  end process;
end vmul8x8p_arch;
```

SIGNALS VS. VARIABLES

Variables are used rather than signals in the process in Table 5-57 to make simulation run faster. Variables are faster because the simulator keeps track of their values only when the process is running. Because variable values are assigned sequentially, the process in Table 5-57 is carefully written to compute values in the proper order. That is, a variable cannot be used until a value has been assigned to it.

Signals, on the other hand, have a value at all times. When a signal value is changed in a process, the simulator schedules a future event in its event list for the value change. If the signal appears on the right-hand side of an assignment statement in the process, then the signal must also be included in the process' sensitivity list. If a signal value changes, the process will then execute again, and keep repeating until all of the signals in the sensitivity list are stable.

In Table 5-57, if you wanted to observe internal values or timing during simulation, you could change all the variables (except `i` and `j`) to signals and include them in the sensitivity list. To make the program syntactically correct, you would also have to move the `type` and `signal` declarations to just after the architecture statement, and change all of the "`:=`" assignments to "`<=`".

Suppose that after making the changes above, you also reversed the order of the indices in the `for` loops (e.g., "`7 downto 0`" instead of "`0 to 7`"). The program would still work. However, dozens of repetitions of the process would be required for each input change in `X` or `Y`, because the signal changes in this circuit propagate from the lowest index to the highest.

While the choice of signals vs. variables affects the speed of simulation, with most VHDL synthesis engines it does not affect the results of synthesis.

and whose carry outputs are 0. The third, nested `for` loop corresponds to the main array of adders in Figure 5-98, all except the last row, which is handled by the fourth `for` loop. The last two `for` loops assign the appropriate adder outputs to the multiplier output signals.

ON THE THRESHOLD OF A DREAM

A three-input "majority function," `MAJ`, is defined at the beginning of Table 5-57 and is subsequently used to compute carry outputs. An *n*-input *majority function* produces a 1 output if the majority of its inputs are 1, two out of three in the case of a 3-input majority function. (If n is even, $n/2+1$ inputs must be 1.)

Over thirty years ago, there was substantial academic interest in a more general class of *n*-input *threshold functions* which produce a 1 output if k or more of their inputs are 1. Besides providing full employment for logic theoreticians, threshold functions could realize many logic functions with a smaller number of elements than could a conventional AND/OR realization. For example, an adder's carry function requires three AND gates and one OR gate, but just one three-input threshold gate.

(Un)fortunately, an economical technology never emerged for threshold gates, and they remain, for now, an academic curiosity.

Table 5-58 Structural VHDL architecture for an 8×8 combinational multiplier.

```
architecture vmul8x8s_arch of vmul8x8s is
component AND2
  port( I0, I1: in STD_LOGIC;
        O: out STD_LOGIC );
end component;
component XOR3
  port( I0, I1, I2: in STD_LOGIC;
        O: out STD_LOGIC );
end component;
component MAJ   -- Majority function, O = I0*I1 + I0*I2 + I1*I2
  port( I0, I1, I2: in STD_LOGIC;
        O: out STD_LOGIC );
end component;

type array8x8 is array (0 to 7) of STD_LOGIC_VECTOR (7 downto 0);
signal PC: array8x8;      -- product-component bits
signal PCS: array8x8;     -- full-adder sum bits
signal PCC: array8x8;     -- full-adder carry output bits
signal RAS, RAC: STD_LOGIC_VECTOR (7 downto 0); -- sum, carry
begin
  g1: for i in 0 to 7 generate    -- product-component bits
    g2: for j in 0 to 7 generate
        U1: AND2 port map (Y(i), X(j), PC(i)(j));
    end generate;
  end generate;
  g3: for j in 0 to 7 generate
    PCS(0)(j) <= PC(0)(j);  -- initialize first-row "virtual" adders
    PCC(0)(j) <= '0';
  end generate;
  g4: for i in 1 to 7 generate   -- do full adders except the last row
    g5: for j in 0 to 6 generate
      U2: XOR3 port map (PC(i)(j),PCS(i-1)(j+1),PCC(i-1)(j),PCS(i)(j));
      U3: MAJ  port map (PC(i)(j),PCS(i-1)(j+1),PCC(i-1)(j),PCC(i)(j));
      PCS(i)(7) <= PC(i)(7); -- leftmost "virtual" adder sum output
    end generate;
  end generate;
  RAC(0) <= '0';
  g6: for i in 0 to 6 generate  -- final ripple adder
    U7: XOR3 port map (PCS(7)(i+1), PCC(7)(i), RAC(i), RAS(i));
    U3: MAJ  port map (PCS(7)(i+1), PCC(7)(i), RAC(i), RAC(i+1));
  end generate;
  g7: for i in 0 to 7 generate
    P(i) <= PCS(i)(0);  -- get first 8 product bits from full-adder sums
  end generate;
  g8: for i in 8 to 14 generate
    P(i) <= RAS(i-8);   -- get next 7 bits from ripple-adder sums
  end generate;
  P(15) <= RAC(7);      -- get last bit from ripple-adder carry
end vmul8x8s_arch;
```

The program in Table 5-57 can be modified to use structural VHDL as shown in Table 5-58. This approach gives the designer complete control over the circuit structure that is synthesized, as might be desired in an ASIC realization. The program assumes that the architectures for `AND2`, `XOR3`, and `MAJ3` have been defined elsewhere, for example, in an ASIC library.

generate statement

This program makes good use of the *generate statement* to create the arrays of components used in the multiplier. The `generate` statement must have a label, and similar to a `for-loop` statement, it specifies an iteration scheme to control the repetition of the enclosed statements. Within `for-generate`, the enclosed statements can include any concurrent statements, `IF-THEN-ELSE` statements, and additional levels of looping constructs. Sometimes `generate` statements are combined with `IF-THEN-ELSE` to produce a kind of conditional compilation capability

Well, we said we'd save the best for last, and here it is. The IEEE `std_logic_arith` library that we introduced in Section 5.9.6 defines multiplication functions for `SIGNED` and `UNSIGNED` types, and overlays these functions onto the "*" operator. Thus, the program in Table 5-59 can multiply unsigned numbers with a simple one-line assignment statement. Within the IEEE library, the multiplication function is defined behaviorally, using the shift-and-add algorithm. We could have showed you this approach at the beginning of this subsection, but then you wouldn't have read the rest of it, would you?

Table 5-59
Truly behavioral VHDL program for an 8×8 combinational multiplier.

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

entity vmul8x8i is
    port (
        X: in UNSIGNED (7 downto 0);
        Y: in UNSIGNED (7 downto 0);
        P: out UNSIGNED (15 downto 0)
    );
end vmul8x8i;

architecture vmul8x8i_arch of vmul8x8i is
begin
  P <= X * Y;
end vmul8x8i_arch;
```

References

Digital designers who want to write better should start by reading the classic *Elements of Style*, 3rd ed., by William Strunk, Jr. and E. B. White (Allyn & Bacon, 1979). Another book on writing style, especially for nerds, is *Effective*